Interactive Abstractions: Proving Safety Properties by Integrating Static Analysis, Theorem Proving and Abstraction

Vlad Rusu, Eli Singerman

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Interactive Abstractions: Proving Safety Properties by Integrating Static Analysis, Theorem Proving and Abstraction

Vlad Rusu IRISA/INRIA Rennes (France)

Eli Singerman SRI International, Menlo Park, California (USA)

N° 3726

Juillet 1999

THÈME 1

ISSN 0249-6399

INRIA
RENNES

Interactive Abstractions: Proving Safety Properties by Integrating Static Analysis, Theorem Proving and Abstraction

Vlad Rusu* IRISA/INRIA Rennes (France)
Eli Singerman† SRI International, Menlo Park, California (USA)

Thème 1 — Réseaux et systèmes
Projet PAMPA

Rapport de recherche n3726 — Juillet 1999 — 21 pages

Abstract: We present a new approach for proving safety properties of reactive systems, based on tight interaction between static analysis, theorem proving and abstraction techniques. The method incrementally constructs a proof or finds a counterexample. Every step consists of applying one of the techniques and makes constructive use of information obtained from failures in previous steps. The amount of user intervention is limited and is highly guided by the system at each step. We demonstrate the method on some examples and show that by using it one can prove more properties than by using each component as a stand-alone.

Key-words: formal verification, static analysis, theorem proving, abstraction.

(Résumé : tsvp)

This research was supported by National Science Foundation grant CCR-9509931. Most of the work was done while the first author was visiting SRI (partially supported by a Lavoisier grant of the French Government). A preliminary version of this paper has appeared in *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'99),LNCS 1579.

* rusu@irisa.fr
† singermn@csl.sri.com

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Preuve de propriétés de sûreté par utilisation conjointe d'analyse statique, preuve de théoremes et abstractions

Résumé : Nous présentons une nouvelle approche pour prouver des propriétés de sûreté de systèmes réactifs. L'approche est basée sur une intéraction forte entre des techniques d'analyse statique, de preuve de théorèmes et d'abstraction. L'approche permet de construire de manière incrémentale une preuve ou un contre-exemple. Chaque pas consiste en l'application de l'une de ces techniques, en utilisant de manière constructive les informations obtenues par l'analyse des échecs des pas précédents. Ainsi, l'intervention de l'utilisateur est limitée, chaque pas étant fortement guidé par le système. Nous présentons l'application de la méthode sur quelques exemples, et montrons qu'elle peut résoudre strictement plus de problèmes que ses composantes utilisées individuellement.

Mots-clé : vérification formelle, analyse statique, preuve de théorèmes, abstractions.

1 Introduction

Theorem proving [GM95, ORS+95, CCF+97][1] is a powerful and general way to verify safety properties of reactive systems, but its use in mechanical verification requires a serious amount of both insightful and labor-intensive manual guidance from the human verifier. Model checking [BCM+92, H91, LPY97] is largely automatic but it only addresses a limited class of essentially finite-state systems. Abstraction [SUM96, DGG97, GS97, BLO98, CU98] can be used to translate an infinite-state system to a finite-state system so as to preserve the property being verified. This can reduce the manual burden of the verification but the discovery of a suitable property-preserving abstraction takes considerable human ingenuity. Furthermore, when the abstracted system fails verification, this could either be because the abstraction was too coarse or because the system did not satisfy the property. It takes deep insight to draw useful information from such a failure. This paper addresses these problems by presenting a methodology for integrating static analysis [CC77, HPR97, BL], theorem proving and abstraction that does not tax the patience and ingenuity of the human verifier. In this methodology, which we call *interactive abstractions*

1. The choice of the abstraction mapping can be guided by the subgoals in a failed proof attempt.

2. A failed verification attempt at the abstract level suggests either auxiliary invariants or a more refined abstraction.

3. The iterative process, when it terminates, yields a proof that the property is satisfied or a counterexample indicating how the property is violated.

We show that our method it is strictly more powerful than verification based on theorem proving with systematic dynamic invariant strengthening techniques.

In interactive abstractions, the verification starts with a one-time use of static analysis generating true-by-construction invariants that are communicated to both the theorem-proving and abstraction components. The rest of the process involves a tight interaction between the prover and abstraction generator, in which each step makes constructive use of information obtained from failures in previous steps. The user is assisted in discovering relevant auxiliary invariants and suitable abstraction mappings while progressing towards a proof or a counterexample. Using this "small increments" approach the required amount of user ingenuity is reduced. Instead of having to rely on keen insight of the problem right from the start, the user gains increasing insight as she progresses in the verification task, enabling her to conclude eventually.

The rest of the paper is organized as follows. In Section 2 we present some basic terminology and an overview of the static analysis, theorem-proving and abstraction

[1] We cite only a few of the relevant contributions in each domain.

techniques that we are using. Section 3 presents the interactive abstractions methodology for integrating these techniques, which we introduce through the verification of a simple example. Section 4 contains a formal comparison of the relative power of the invariant-strengthening and boolean abstraction techniques, together with an example demonstrating that interactive abstractions is strictly more powerful than each of these methods. We conclude in Section 5 and present some future work directions.

2 The Components

2.1 Terminology

We use transition systems as a computational model for reactive programs. A transition system T consists of a finite set of typed variables V, an initial condition Θ and a finite set of guarded transitions $\mathcal{T}$. The variables can be either control or data variables. The control variables are of a finite type *Location*. Each transition $\tau \in \mathcal{T}$ is labeled and composed of a guard and an assignment. For example, in the transition system illustrated in Fig. 1, the type *Location* consists of the two values l_1, l_2. There are two transitions, $\tau_1 : \quad l_1 \stackrel{inc}{\rightarrow} l_2$ and $\tau_2 : \quad l_2 \stackrel{dec}{\rightarrow} l_1$. We suppose that transitions are uniquely determined by their origin, destination and label. Whenever there is no risk of confusion we refer to transitions by their label, e.g, the transition *inc* in Fig. 1.

A state is a type-consistent valuation of the variables. The initial condition Θ is a predicate on states. Each transition τ induces (through its guard and assignment) a transition relation ρ_τ relating the possible before- and after-states. The global transition relation of the system is $\rho_T = \bigcup_{\tau \in \mathcal{T}} \rho_\tau$. A computation of the transition system is an infinite sequence of states, in which the first state satisfies the initial condition and every two consecutive states are in the transition relation. Computations are obtained by "firing" (or "taking") consecutive transitions. For example, in the transition system illustrated in Fig. 1, the initial state has location l_1 and variable $x = 0$. From this state the program can take the transition *inc* (its guard is *true*), assigning 1 to x, moving to location l_2, etc.

The parallel (asynchronous) composition of transition systems is defined via interleaving in the usual manner. For a transition τ and a state predicate φ, the predicate $pre_\tau(\varphi)$ characterizes all the states that have a τ-successor in which φ holds:

$$pre_\tau(\varphi): \quad \exists s'.\rho_\tau(s, s') \wedge \varphi(s').$$

Likewise, the predicate $\widetilde{pre}_\tau(\varphi)$ characterizes all the states from which, after taking transition τ, φ holds:

$$\widetilde{pre}_\tau(\varphi): \quad \forall s'.\rho_\tau(s, s') \supset \varphi(s').$$

The predicate $post_\tau(\varphi)$ characterizes the states that can be reached by taking transition τ from some state satisfying φ:

$$post_\tau(\varphi): \quad \exists s.\rho_\tau(s, s') \wedge \varphi(s).$$

These predicates are also defined globally for the whole transition system T as: $pre_T(\varphi)$: $\exists s'.\rho_T(s, s') \wedge \varphi(s')$, $\widetilde{pre}_T(\varphi)$: $\forall s'.\rho_T(s, s') \supset \varphi(s')$, $post_T(\varphi)$: $\exists s.\rho_T(s, s') \wedge \varphi(s)$. In the sequel, we omit T when it is understood from the context.

We now briefly describe the static analysis, theorem proving and abstraction techniques that we integrate in *interactive abstractions*. In should be stressed that the choice of the particular tools that we use is not the main point here, but rather the way in which we integrate them. For example, one could use POLKA [HPR97] as the static analysis tool, INVEST [BLO98] as the abstraction tool, etc.

2.2 Static Analysis

For automatically generating invariants we use a method similar to that suggested by [BL]. The analysis starts by computing local invariants at every control location. The local invariant of a control location l is the disjunction of $post_\tau(true)$, for all transitions τ leading to l. Then, the local invariants are propagated to other control locations of the system to obtain global invariants. For example, in the transition system illustrated in Fig. 1, static analysis yields the local invariant $\Box(pc = l_1 \supset x \geq 0)$. Since $x \geq 0$ is preserved by transition *inc* and is true initially it is a global invariant.

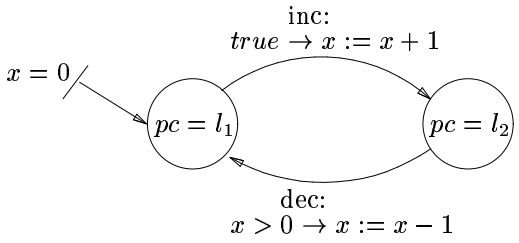

Figure 1: Example of transition system.

2.3 Theorem Proving

We use PVS [ORS+95] for invariant strengthening [GS96, HS96]. Given a transition system T and a state predicate I we say that I is *inductive* if $I \supset \widetilde{pre}(I)$. Obviously, if I is inductive and holds at the initial states of T then I is an invariant of T. When I is not inductive we can strengthen it by taking $I \wedge \widetilde{pre}(I)$ and check if the latter

is inductive, i.e., $I \wedge \widetilde{pre}(I) \supset \widetilde{pre}(I \wedge \widetilde{pre}(I))$ or equivalently $I \wedge \widetilde{pre}(I) \supset \widetilde{pre}^2(I)$. In general, this procedure terminates if there exists an n such that

$$I \wedge \widetilde{pre}(I) \wedge \ldots \wedge \widetilde{pre}^n(I) \supset \widetilde{pre}^{n+1}(I). \quad (1)$$

If this is the case, it follows that $I \wedge \widetilde{pre}(I) \wedge \ldots \wedge \widetilde{pre}^n(I)$ is inductive, and if I also holds initially, then it is an invariant.

This technique can be implemented in PVS as follows. We use a simple invariance rule stating that I is an invariant if it is true initially and is preserved by all transitions. If I is inductive then applying the rule once would complete the proof. Otherwise, the prover presents a number of pending (unproved) subgoals. Each subgoal results from the fact that I is not preserved by some transition. We then apply the invariance rule to the predicate obtained by taking the conjunction of I and all the unproved subgoals. This amounts to attempting to prove that $I \wedge \widetilde{pre}(I)$ is inductive. If there exists an n such that (1) holds then repeating this process n times would eliminate all the subgoals and complete the proof. This leads to a fully automatic procedure (that is not guaranteed to terminate).

2.4 Abstraction

We use the boolean abstraction technique described in [GS97]. The abstraction of a concrete transition system T relative to a finite set of state predicates $\mathcal{B} = \{B_1, \ldots, B_k\}$ called *boolean abstract variables* is a transition system denoted $T/\mathcal{B}$. The states of the abstract system $T/\mathcal{B}$ are called *abstract states*. Each abstract state is labeled with a valuation of the control variables of T and of the boolean abstract variables. Let us now briefly describe how $T/\mathcal{B}$ is constructed.

The initial abstract state is labeled with the initial control configuration of T and with the truth values of the boolean abstract variables at the initial concrete state. Assume now that s^A is an abstract state, the abstract transitions going out of s^A are computed as follows. Every concrete transition τ originating from a concrete state with the same control configuration as s^A can give rise to several abstract transitions. Each of these transitions has the same label as τ and leads to an abstract state obtained by computing (with PVS) the effect of τ, starting from s^A, on the control and boolean abstract variables.

Consider, for example, the concrete system illustrated in Fig. 2(a). An abstraction relative to $B_1 : (x = 0)$ and $B_2 : (x = 1)$ generates the abstract system represented in Fig. 2(b). The abstraction only relative to B_2 yields the abstract system represented in Fig. 2(c) (of which only the initial portion is shown). Note that in the latter, simulating the concrete transition *inc* gives rise to two successors, s_1^A and s_2^A. This is because starting at the initial abstract state s_0^A, where $\neg(x = 1)$ holds, performing $x := x + 1$ can either lead to a state in which $(x = 1)$ is true or to a state in which the latter is false. Note also that in the abstract system represented

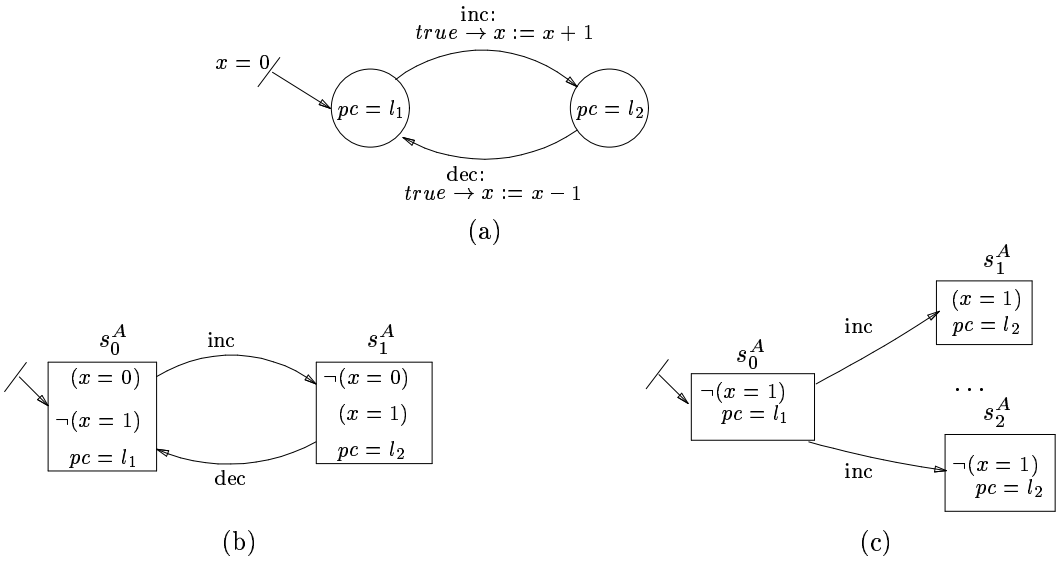

Figure 2: (a) concrete system (b), (c) abstract systems.

in Fig. 2(b) the only abstract state labeled $pc = l_2$ is s_1^A and this state also labeled $(x = 1)$. We say that abstraction (b) "shows" the property $\Box(pc = l_2 \supset (x = 1))$. On the other hand, the abstraction (c) does not show this property, since there exists the abstract state s_2^A labeled $pc = l_2$ and $\neg(x = 1)$.
To define the notion of "abstraction showing a property" we interpret the labelling of each abstract state s^A as a predicate $\pi(s^A)$ on the concrete variables. For example, the predicate associated with the initial state of system (c) above is $(pc = l_1) \wedge \neg(x = 1)$. Let $T/\mathcal{B}$ be the abstraction of a concrete system T relative to the set of abstract variables $\mathcal{B} = \{B_1, \ldots, B_k\}$ and φ be a state predicate. We say that an abstract state s^A *shows* φ if $\pi(s^A)$ implies φ. We say that $T/\mathcal{B}$ shows $\Box\varphi$, denoted $T/\mathcal{B} \models_{ABS} \Box\varphi$, if all abstract states show φ. The crucial feature of these boolean abstractions, which is true by construction, is that for each computation

$$\sigma: \quad s_0 \stackrel{\alpha_0}{\to} s_1 \stackrel{\alpha_1}{\to} \cdots s_{n-1} \stackrel{\alpha_{n-1}}{\to} s_n \tag{2}$$

of the concrete system T, there exists a trace

$$\sigma^A: \quad s_0^A \stackrel{\alpha_0}{\to} s_1^A \stackrel{\alpha_1}{\to} \cdots s_{n-1}^A \stackrel{\alpha_{n-1}}{\to} s_n^A \tag{3}$$

of the abstract system $T/\mathcal{B}$, such that for $i = 0, \ldots, n$ the labels of the abstract and concrete transitions coincide, and both the values of the boolean abstract variables and the values of the control variables in s_i^A and in s_i coincide. Consequently,

boolean abstractions are useful for proving invariants, since

$$T/\mathcal{B} \models_{ABS} \Box\varphi \quad \Rightarrow \quad T \models \Box\varphi.$$

In general, an abstraction relative to a larger set of abstract variables can "show" more properties, because the prover has more information at its disposal when new abstract states are generated, therefore it can eliminate some of them, yielding a finer abstraction. Also, constructing an abstraction with some known invariants of the concrete system can assist in eliminating irrelevant abstract states.

2.5 Trace Analysis

Even though every concrete trace can be matched by an abstract one, the converse is not always true. Thus, when a safety property is violated at the abstract level, it does not necessarily follow that the property is violated at the concrete level as well. For example, in the abstract system represented in (Fig. 2(c)) the abstract trace $s_0^A \stackrel{inc}{\rightarrow} s_2^A$ violates the property $\Box(pc = l_2 \supset x = 1)$. Nevertheless, the concrete system (a) does satisfy this property. It is therefore essential to carefully analyze an "abstract counterexample" in order to decide whether it indicates the existence of a "concrete counterexample".

In *interactive abstractions*, when an abstract trace violates the property under consideration, we use forward analysis to decide if it can be "matched" by a concrete computation. If true, the property is violated in the concrete system. If not, this indicates that the abstract system is too coarse. We then use backward analysis to derive useful information that would help in refining further abstractions.

Before describing our forward and backward analysis techniques in more detail, let us formalize some notions.

1. Let s^A be an abstract state and φ a state property. We say that s^A *violates* φ, if the conjunction $\pi(s^A) \wedge \neg\varphi$ is satisfiable. Here, $\pi(s^A)$ is the predicate labelling s^A (cf. Section 2.4). For example, the abstract state s_2^A (Fig. 2(c)) violates $(pc = l_2 \supset x = 1)$.

2. An abstract trace

$$\sigma^A : \quad s_0^A \stackrel{\alpha_0}{\rightarrow} s_1^A \stackrel{\alpha_1}{\rightarrow} \cdots s_{n-1}^A \stackrel{\alpha_{n-1}}{\rightarrow} s_n^A$$

is called an *abstract counterexample* for a safety property $\Box\varphi$, if its last state, s_n^A, violates φ.

2.5.1 Forward Analysis

Assume now that σ^A is an abstract counterexample for a safety property $\Box\varphi$. Intuitively, checking whether σ^A gives rise to a concrete counterexample can be done

as follows. We symbolically simulate the abstract trace on the concrete system. At each step, the boolean value of the abstract variables is compared between the abstract and concrete levels. If the simulation was successful (i.e., at each step the corresponding transition was enabled on the concrete system and the boolean values matched between the concrete and abstract levels) then we say the abstract trace σ^A can be *matched* with a concrete computation. To check whether the latter is a concrete counterexample, we have to check whether it violates the property $\Box\varphi$. This can also be done using the symbolic simulation.

Formally, let $\sigma^A : \quad s_0^A \xrightarrow{\alpha_0} s_1^A \xrightarrow{\alpha_1} \cdots s_{n-1}^A \xrightarrow{\alpha_{n-1}} s_n^A$ be an abstract trace. For relating abstract transitions to concrete ones, note that, for every abstract transition $\tau_i^A : \quad s_i^A \xrightarrow{\alpha_i} s_{i+1}^A$ $(i = 0 \ldots n-1)$, there exists a unique concrete transition $\tau_i : l_i \xrightarrow{\alpha_i} l_{i+1}$, such that location l_i labels s_i^A and l_{i+1} labels s_{i+1}^A. Here, τ_i is the concrete transition that generated τ_i^A when the abstract system was built (cf. Section 2.4).

The *post-image of the initial condition* Θ *on* σ^A, denoted $post_{\sigma^A}(\Theta)$ is defined as:

$$post_{\sigma^A}(\Theta) = \pi(s_n^A) \wedge post_{\tau_{n-1}}(\pi(s_{n-1}^A) \wedge \cdots post_{\tau_1}(\pi(s_1^A) \wedge post_{\tau_0}(\pi(s_0^A) \wedge \Theta))) \quad (4)$$

That is, $post_{\sigma^A}(\Theta)$ is obtained by successively computing the *post* of transitions τ_0 ... τ_{n-1}, starting with the predicate Θ, where at each step the result is conjoined with the predicate labelling the current abstract state. The following criterion allows to decide whether there exists a concrete counterexample matching an abstract one.

Proposition 2.1 *Let σ^A be an abstract trace and φ a state predicate such that σ^A is an abstract counterexample for $\Box\varphi$. Then, there exists a concrete counterexample for $\Box\varphi$ matching σ^A if and only the predicate $post_{\sigma^A}(\Theta) \wedge \neg\varphi$ is satisfiable.*

For example, consider again the abstract system represented in Fig. 2(c). We have shown that the trace $s_0^A \xrightarrow{inc} s_2^A$ is an abstract counterexample for the property $\Box(pc = l_2 \supset x = 1)$. Let us show that it is not a concrete one. We simulate this trace on the concrete system (Fig. 2(a)). By computing $post_{inc}(\ (\pi(s_0^A) : \quad pc = l_1 \wedge \neg(x = 1)) \wedge (\Theta : \quad pc = l_1 \wedge (x = 0))\ )$ we obtain the predicate $(pc = l_2 \wedge x = 1)$. The conjunction of the latter with $\pi(s_2^A) : \quad (pc = l_2 \wedge \neg(x = 1))$ is unsatisfiable. Thus, there is no concrete counterexample matching this abstract counterexample.

Note that here, we did not need to go through all the process for being able to conclude (that would require also taking the conjunction with $\neg\varphi$) because an unsatisfiable predicate was found "earlier" in the process. This demonstrates an advantage of using the criterion defined by Proposition 2.1: it is often possible to rule-out an abstract counterexample without computing all of the predicate (4).

2.5.2 Backward Analysis

When an abstract counterexample cannot be matched by a concrete one, this is an indication that the abstraction is "too coarse". In interactive abstractions, we

employ the trace analyzer to obtain information on why this has happenned. Using this feedback it is usually possible to progress in the proof.

We demonstrate the use of backward analysis by a simple scenario. Suppose we want to prove, using abstractions, that the system in Fig. 2(a) satisfies the safety property $\Box(pc = 2 \supset (x = 1))$. If we start by constructing the abstract system represented in Fig. 2(c), then forward analysis (2.5.1) reveals that the trace $s_0^A \stackrel{inc}{\rightarrow} s_2^A$ is an abstract counterexample that cannot be matched by a concrete one. Thus, the abstract state $(pc = l_2 \wedge \neg(x = 1))$ is not "reachable" in the concrete system. To find out why this happened we backtrack the violating abstract state along the sequence $l_1 \stackrel{inc}{\rightarrow} l_2$ (that generated the abstract counterexample). Backtracking is done by computing, on the concrete system, *pre*-images of the violating abstract state over the given sequence. We compute $pre_{inc}(pc = l_2 \wedge \neg(x = 1))$ and obtain $(pc = l_1 \wedge \neg(x = 0))$. This predicate is interpreted as follows: the system represented in Fig. 2(c) "believes" that when $pc = l_1$, the predicate $(x = 0)$ can be *false*. This is indeed possible in the abstract system (c), but it is clearly not possible in the concrete system (a). We identify from the feedback the predicate $(x = 0)$ and refine the abstraction (c) by including $(x = 0)$ as a new boolean abstract variable. This yields the the refined abstract system (b) which shows the property.

3 The Integration: Interactive Abstractions

Our interactive abstractions method involves tight integration of the above static analysis, theorem proving, abstraction and trace analysis techniques.

The general scheme is presented in Fig. 3. The verification starts with a one-time use of the invariant generator, that performs static analysis and produces true-by-construction invariants that are communicated to both the theorem prover and abstraction generator. The next step is to employ the theorem prover in an attempt to prove that the property is inductive. If this does not succeed (usually because at this stage the prover does not have enough information to conclude) then a number of unproved subgoals are left pending. The following step is to use the pending subgoals to either remain in the prover and do invariant strenghtening, or to compute an abstraction (using predicates from the pending subgoals as boolean abstract variables). If the abstraction still does not succeed to prove the property, trace analysis is employed to find out whether this happened because the property is not true, or because the abstraction is too coarse. In the latter case, trace analysis also provides feedback information, allowing to either refine the abstraction or to obtain new conjectures that, if proved, become useful auxiliary invariants.

We demonstrate the approach on an alternating-bit protocol illustrated in Fig 4. The transition-system model is taken from [GS97]. There are three processes: sender, receiver and environment. The sender generates messages, records them in the *sent* list, then sends them to the receiver over the communication medium *mes-*

sage_channel. The latter is modeled as a one-place buffer that can hold a message and a bit. The receiver records successfully received messages in the *received* list and sends an acknowledgment through the one-place buffer *ack_channel*. The environment can lose messages and acknowledgements by setting the boolean flags *message_present* and *ack_present* to *false*. This causes the sender/receiver respectively to retransmit. The safety property to be proved is that the lists *sent* and *received* always differ by at most one message. This amounts to proving $\Box I$, where

$$I: \quad sent = received \vee sent = tail(received). \tag{5}$$

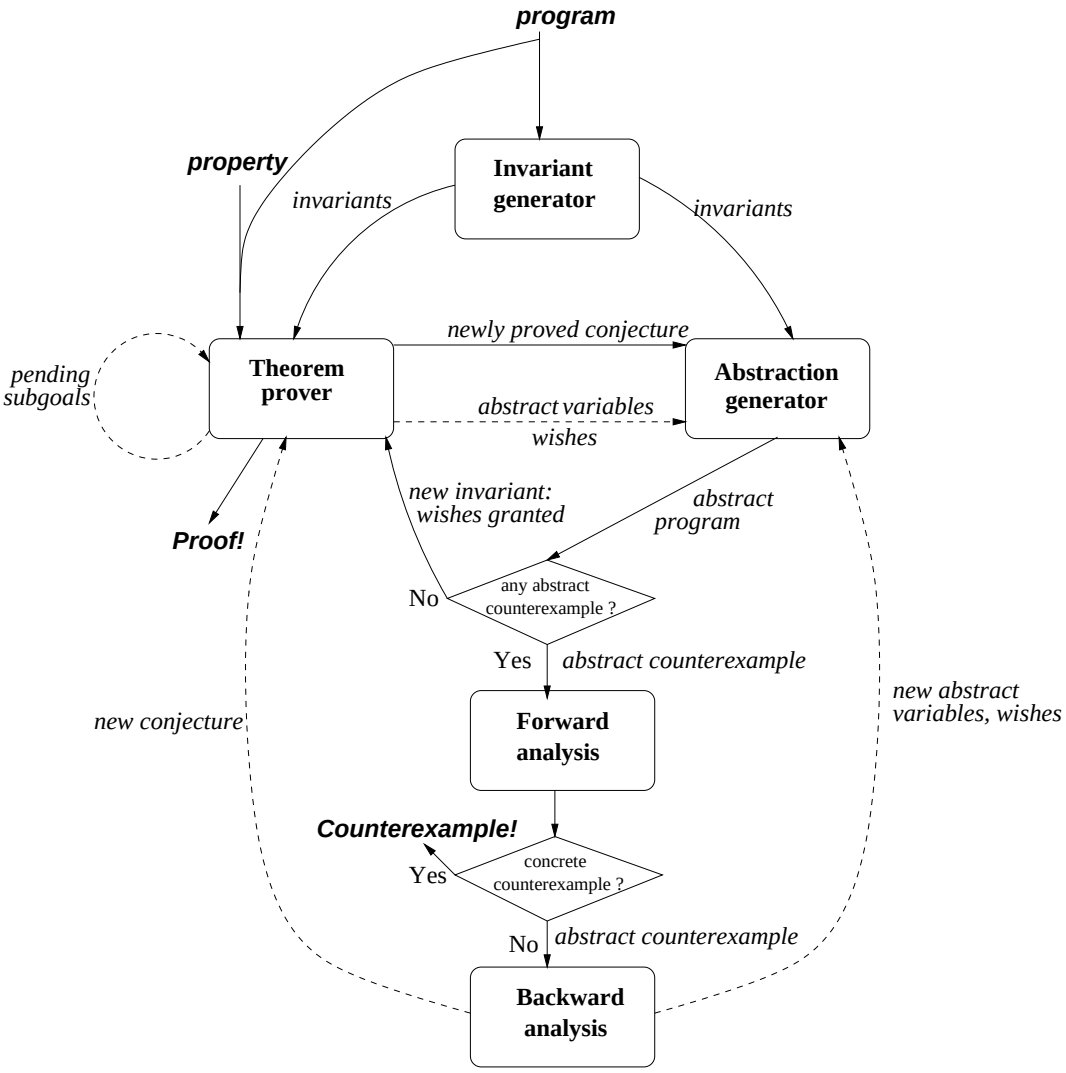

Figure 3: Interactive Abstractions

The first step, static analysis, yields two invariants (not shown here) that are fed to the prover and to the abstraction generator. The next step is theorem proving, and since the property is not inductive, the proof is not completed.

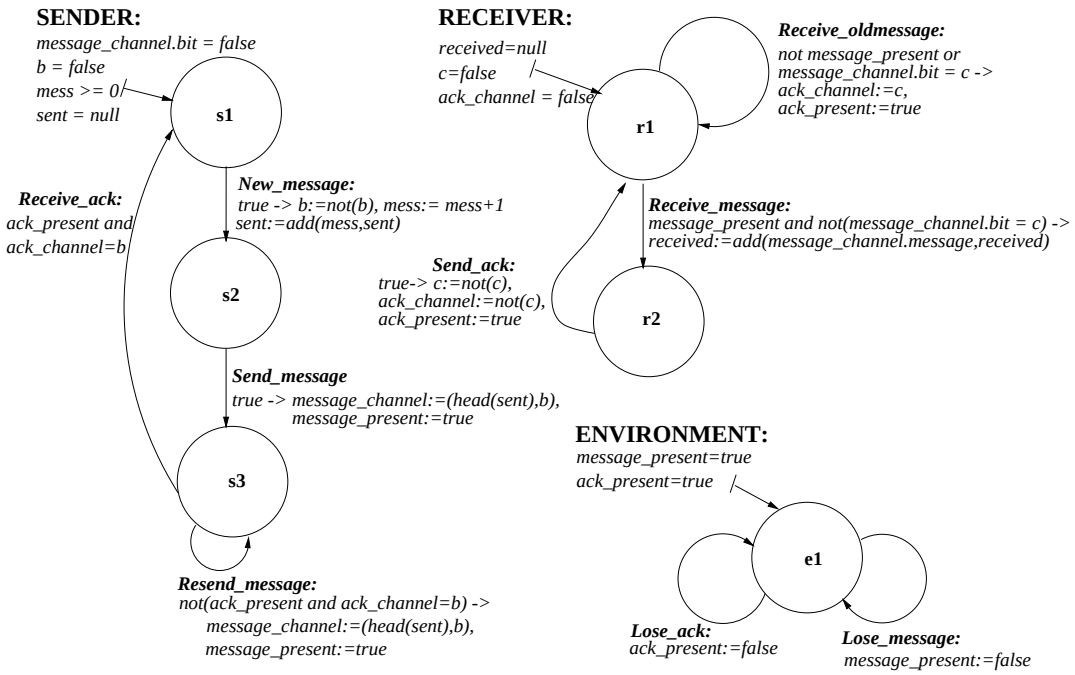

Figure 4: An alternating bit protocol

There are three pending subgoals, all of which are related to transitions that update the *sent*/*received* lists. For example, PVS returns the pending subgoal represented in Fig. 5. (We recall that in PVS, proving a subgoal means proving that the conjunction of assertions "above" the line implies the disjunction of assertions "below" the line.)

$$\frac{\begin{array}{l} sent = received \\ pc_{rec} = r_1 \\ pc_{rec}^{'} = r_2 \end{array}}{sent^{'} = received^{'}}$$

Figure 5: A pending subgoal returned by PVS.

The variable pc_{rec} is the control variable of the receiver process. The primed variables refer to the variables "after" the next transition is taken. Here, the "next transition" is the one labeled ***receive_message*** of the receiver process (cf. Fig. 4),

as indicated by assertions $pc_{rec} = r_1$, $pc'_{rec} = r_2$ in the subgoal. Therefore, the latter can be interpreted as follows: prove that if relation *sent = received* holds before the ***receive_message*** transition is taken, then the same relation holds after the transition is taken. But by examining the transition ***receive_message***, we see that it modifies the variable *received* and does not modify the variable *sent*. Therefore, the only way to settle the given subgoal is to show that the hypotheses (assertions "above" the line) are inconsistent, i.e., before a ***receive_message*** transition is taken, the relation *sent = received* cannot hold. This amounts to proving $\Box\varphi_1$, where φ_1 is the predicate $(pc_{rec} = r_1) \wedge (pc'_{rec} = r_2) \supset \neg(sent = received)$.

The second pending subgoal (not shown here) is related to the ***new_message*** transition. It is symmetric to the subgoal in Fig. 5 and can be settled by proving $\Box\varphi_2$, where $\varphi_2 : \quad (pc_{send} = s_1) \wedge (pc'_{send} = s_2) \supset \neg(sent = tail(received))$.

The third pending subgoal is related to the transition ***receive_message***. It requires to prove that whenever a message is received and appended to the *received* list, it is equal to the message in head of the *sent* list. This amounts to proving $\Box\varphi_2$, with $\varphi_2 : \quad (pc_{rec} = r_1) \wedge (pc'_{rec} = r_2) \supset (message\_channel.message = head(sent))$.

We now have have two options[2] to continue:

- either to stay in the theorem prover and try to prove that the assertion $I \wedge \varphi_1 \wedge \varphi_2 \wedge \varphi_3$ is inductive (property I was defined by equation (5))
- or try to show $\Box\varphi_1$, $\Box\varphi_2$, $\Box\varphi_3$ by using the abstraction generator.

We choose the latter option. To construct an abstraction, we identify from the pending subgoals the three predicates *sent = received*, *sent = tail(received)* and *message_channel.message = head(sent)* as potentially relevant abstract variables and communicate them to the abstraction generator, together with three corresponding *wishes*. A wish is a property of the abstract system that, if true, would allow to settle a pending subgoal. Here, the wishes are that properties $\Box\varphi_1$, $\Box\varphi_2$, $\Box\varphi_3$ hold on the abstract system. The first property signifies that, in the abstract system, there are no transitions labelled ***receive_message***, starting from an abstract state labelled $pc_{rec} = r_1 \wedge (sent = received)$ and leading to an abstract state labelled $pc_{rec} = r_2 \wedge (sent = received)$. The properties $\Box\varphi_2$, $\Box\varphi_3$ also signify the absence of some specific transitions in the abstract system.

Next, the abstraction generator automatically computes an abstract system relative to the given boolean abstract variables. The resulting system does contain an abstract counterexample for $\Box\varphi_1$, meaning that the corresponding wish is not granted and trace analysis (cf. Section 2.5) is necessary.

The abstract counterexample for $\Box\varphi_1$ is a trace consisting of one transition labelled ***receive_message***, starting from the initial abstract state. The next step

[2] user-dependent choices are represented in *interactive abstractions* as dashed arrows (cf. Fig. 3).

is to check whether the abstract counterexample can be matched by a concrete computation (in the sense defined in Section 2.5.1). The forward analysis component (Fig. 3) reveals that this is not the case, because transition ***receive_message*** is not enabled in the initial concrete state. Therefore, the abstraction is "too coarse". The following step is to use backward analysis to find out why this happenned. In our case, there is no need to backtrack because the violating abstract state for φ_1 is the initial abstract state. We just evaluate in the initial *concrete* state the guard of transition ***receive_message***. This identifies the cause: the conjunct $message\_channel.bit = c$ from the guard, which evaluates to *true* initially (cf. Fig. 4) and disables the transition. The current abstract system "knows nothing" about this predicate, this is why it "believes" that ***receive_message*** is enabled initiallly.

The obvious choice now is to take $message\_channel.bit = c$ as a new abstract variable and to redo an abstraction. Still, the second abstraction does not grant our wishes, since it contains abstract counterexample for $\Box\varphi_2$. The latter is the sequence of transitions ***new_message***, ***send_message***, ***receive_ack***. Again, forward analysis reveals that the abstract counterexample cannot be matched by a concrete one, because the transition ***receive_ack*** in the sequence is not enabled at the concrete level. Now, backward analysis of this transition's guard to the initial configuration reveals the cause: the conjunct $ack\_channel = b$ in the guard, which is unknown to the current abstraction.

We add the discovered predicates $message\_channel.bit = c$ and $ack\_channel = b$ to the list of abstract variables and let the abstraction generator build a new abstraction. Finally, all the wishes are granted, which settles the pending subgoals and allows the prover to formally complete the proof.

In [GS97] the same protocol is analyzed by an abstraction relative to an a-priori chosen set of sub-formulas of the guards. Our approach allows to *discover* the relevant abstract variables, which are suggested to the user by the failures of previous attempts. The method can be automated in significant proportion: all the components in the diagram (Fig. 3) perform automatic tasks, and user intervention is limited to choosing between the abstraction generator and theorem prover components. In both cases, the user is assisted in providing the relevant abstract variables by the pending subgoals in the prover and by the feedback information obtained by trace analysis. The method is incremental: progress is made in each step, as refined abstractions reduce the search space, and the user gains insight of the problem while progressing towards a proof or a counterexample.

4 Integration is More Powerful

In this section we take a closer look at the theorem prover and abstraction generator components and compare their relative power for proving safety properties. We

demonstrate on a simple example that *interactive abstractions*, which integrates these fully-automatic components with a limited amount of user intervention (highly guided by feedback from the forward/backward analysis of traces) is strictly more powerful than its components used as stand-alones.

We define the class of safety properties that can be proved by the theorem prover using the invariant-strengthening technique described in Section 2.3. For a transition system T and $n = 0, 1, \ldots$ consider the set

$$INV_n(T) = \{\Box I \mid I \wedge \widetilde{pre}(I) \wedge \ldots \wedge \widetilde{pre}^n(I) \supset \widetilde{pre}^{n+1}(I)\} \tag{6}$$

Definition 4.1 (safety properties provable by $\widetilde{pre}$-invariant strengthening) *The class $INV(T)$ of safety properties that are provable by $\widetilde{pre}$-invariant strengthening is $\bigcup_{n\geq 0} INV_n(T)$.*

Next, we define the class of safety properties that can be proved by the abstraction generator component (described in Section 2.4). For a state predicate I and $n = 0, 1, \ldots,$ let

$$AV_n(I) \;=\; \{I, \widetilde{pre}(I), \ldots, \widetilde{pre}^n(I)\}.$$

The elements of $AV_n(I)$ are meant to be used as potential abstract variables. It should be stressed that this choice is not arbitrary. In fact, in *interactive abstractions*, the abstract variables suggested by the trace analysis (such as the guards of the transitions) are elements of $AV_n(I)$ for some n. The set $ABS_n(T)$ of safety properties that can be shown by abstraction relative to a subset of $AV_n(I)$ is

$$ABS_n(T) \;=\; \{\Box I \mid \exists \mathcal{B} \subseteq AV_n(I) \; s.t. \; T/\mathcal{B} \models_{ABS} \Box I\}.$$

Definition 4.2 (safety properties that can be shown by $\widetilde{pre}$-abstraction) *The class $ABS(T)$ of safety properties that can be shown by $\widetilde{pre}$-abstraction is $\bigcup_{n\geq 0} ABS_n(T)$.*

Note that both $\widetilde{pre}$-invariant strengthening and $\widetilde{pre}$-abstraction are fully automatic techniques. Under the assumption that the same "reasoning power" is used for both $\widetilde{pre}$-invariant strengthening and $\widetilde{pre}$-abstraction (for example, both use the same theorem prover), the following holds.

Lemma 4.3 *For every transition system T*

$$ABS_n(T) = INV_n(T) \quad n = 0, 1, \ldots \tag{7}$$

Proof. $ABS_n(T) \subseteq INV_n(T)$: let I be a state predicate such that $\Box I \in ABS_n(T)$. Then, there exists a set $\mathcal{B} \subseteq AV_n(I)$ of abstract variables such that $T/\mathcal{B} \models_{ABS} \Box I$. Since adding abstract variables yields a finer abstraction, we have $T/AV_n(I) \models_{ABS} \Box I$.

Claim. In every state of the abstract system $T/AV_n(I)$, all abstract variables are labelled positively.

proof (claim). Assume by contradiction that s^A is an abstract state in which the abstract variable $\widetilde{pre}^k(I) \in \{I, \widetilde{pre}(I), \ldots, \widetilde{pre}^n(I)\}$ is labelled negatively and k is minimal. If $k = 0$, then s^A is labelled $\neg I$, but this contradicts $T/AV_n(I) \models_{ABS} \Box I$. We are left with the case $k > 0$. Using $\widetilde{pre}^k(I) = \widetilde{pre}(\widetilde{pre}^{k-1}(I))$ and $\neg\widetilde{pre}(\varphi) = pre(\neg\varphi)$ (that are easy to infer from the definitions of the operators in Section 2.1) we obtain $\neg\widetilde{pre}^k(I) = pre(\neg\widetilde{pre}^{k-1}(I))$. Therefore, $pre(\neg\widetilde{pre}^{k-1}(I))$ is also true in the abstract state s^A. Now, by construction of the abstract system (cf. Section 2.4) the successor abstract states of s^A are built using the theorem prover to simulate forward the concrete transitions, starting from s^A. (This amounts to checking the satisfiability of the *post*-images of predicates that are known true in s^A.) Knowing that $pre(\neg\widetilde{pre}^{k-1}(I))$ is true in s^A, the prover will find, by forward simulating some concrete transition τ, that the predicate $\neg\widetilde{pre}^{k-1}(I)$ is satisfiable[3]. Therefore, by simulating some concrete transition τ, the prover will give rise to a successor abstract state of s^A, in which $\widetilde{pre}^{k-1}(I)$ is labelled negatively. This contradicts the minimality of k, settles the case $k > 0$, and completes the proof of the claim.

We know now that in every state of $T/AV_n(I)$, all the abstract variables are labelled positively. By construction of the abstract system, this means that for every concrete transition τ

$$post_\tau(I \wedge \widetilde{pre}(I) \wedge \ldots \wedge \widetilde{pre}^n(I)) \supset I \wedge \widetilde{pre}(I) \wedge \ldots \wedge \widetilde{pre}^n(I)$$

is provable. From this it follows that

$$post(I \wedge \widetilde{pre}(I) \wedge \ldots \wedge \widetilde{pre}^n(I)) \supset \widetilde{pre}^n(I)$$

is provable. From the latter it follows that $I \wedge \widetilde{pre}(I) \wedge \ldots \wedge \widetilde{pre}^n(I) \supset \widetilde{pre}^{n+1}(I)$ is provable. Thus, $\Box I \in INV_n$.

$INV_n(T) \subseteq ABS_n(T)$: let I be a state predicate such that $\Box I \in INV_n(T)$. Thus

$$I \wedge \widetilde{pre}(I) \wedge \ldots \wedge \widetilde{pre}^n(I) \supset \widetilde{pre}^{n+1}(I)$$

is provable. But this is equivalent to the fact that

$$post(I \wedge \widetilde{pre}(I) \wedge \ldots \wedge \widetilde{pre}^n(I)) \supset \widetilde{pre}^n(I)$$

is provable, or, equivalently, that for all concrete transitions τ:

$$post_\tau(I \wedge \widetilde{pre}(I) \wedge \ldots \wedge \widetilde{pre}^n(I)) \supset \widetilde{pre}^n(I)$$

[3]We suppose the prover is powerful enough to handle automatically the operators *post*, *pre*, $\widetilde{pre}$ and the relations between them. Pvs can do this for boolean, linear arithmetic, and list types.

is provable. Now, for all $1 \leq k \leq n$, it is easy to show that $post_\tau(\widetilde{pre}^k(I)) \supset \widetilde{pre}^{k-1}(I)$. Adding more hypotheses does not reduce the proving capabilities, therefore we obtain that for all $1 \leq k \leq n$, the implication $post_\tau(I \wedge \widetilde{pre}(I) \wedge \ldots \wedge \widetilde{pre}^n(I)) \supset \widetilde{pre}^{k-1}(I)$ is provable. We already know that the latter is provable for $k = n+1$. Therefore we conclude that for every concrete transition τ, $post_\tau(I \wedge \widetilde{pre}(I) \wedge \ldots \wedge \widetilde{pre}^n(I)) \supset I \wedge \widetilde{pre}(I) \wedge \ldots \wedge \widetilde{pre}^n(I)$ is provable. So that if we build the abstraction $T/AV_n(I)$ using the same prover to simulate the concrete transitions, we get an abstract system in which the abstract variables $I, \widetilde{pre}(I), \ldots, \widetilde{pre}^n(I)$ are labeled positively in each state. In particular, all abstract state are labeled I, thus $T/AV_n(I) \models_{ABS} \Box I$. This shows the inclusion $INV_n(T) \subseteq ABS_n(T)$ and completes the proof. □

Using the above lemma together with definitions 4.1, 4.2 we obtain the following.

Theorem 4.4 *A safety property can be proved by $\widetilde{pre}$-invariant strengthening iff it can be shown by $\widetilde{pre}$-abstraction.*

We show in the following example that *interactive abstractions* is strictly more powerful than the fully automatic techniques of $\widetilde{pre}$-invariant strengthening and of $\widetilde{pre}$-abstractions. The example is a mutual-exclusion algorithm taken from [BGP97] and is based on the same principle as the well-known Bakery Algorithm: using "tickets" to control access to the critical section. The program is illustrated in Fig. 6: two global variables t_1 and t_2 are used for keeping record of ticket values, and two local variables a and b control the entry to the critical sections. The mutual-exclusion property is formulated as $\Box I$ where

$$I : \neg(pc_1 = cs \wedge pc_2 = cs). \tag{8}$$

We employ *interactive abstractions* to prove this property. First, static analysis generates the local invariants $pc_1 = cs \supset a \leq t_1$ and $pc_2 = cs \supset b \leq t_1$, which are then passed to the theorem prover and to the abstraction generator. Then, theorem proving yields two unproved subgoals, from which we identify the predicates $(a \leq t_1)$ and $(b \leq t_1)$ as relevant abstract variables (note that these predicates are simply the guards). The wish associated with the transition *in-a* is: "any abstract state labeled $pc_1 = nc,\ pc_2 = cs$ is also labeled $\neg(a \leq t_1)$ ". That is, the guard $(a \leq t_1)$ should prevent the first process from entering its critical section while the second is already there. A similar wish is associated with the transition *in-b*.

An abstraction relative $(a \leq t_1)$ and $(b \leq t_1)$ to does not grant these wishes. An abstract counterxample is produced, which is identified by forward analysis as not corresponding to a concrete computation; thus, the abstraction is too coarse. By backtracking the violating abstract state to the initial control configuration, the backward analysis component reveals that the error occurred since the abstraction "believes" that the following predicate: $t_1 \leq t_2 - 1$ can hold when $pc_1, pc_2 = init$.

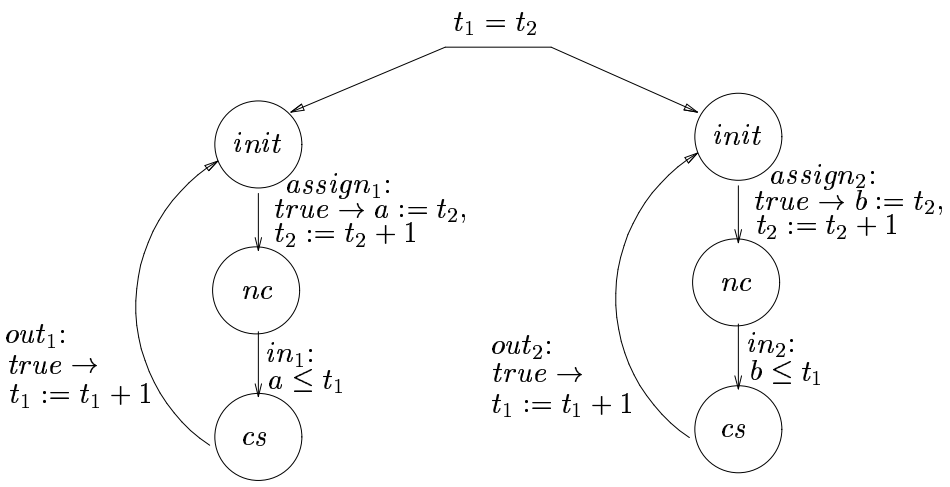

Figure 6: The Ticket Protocol.

The user now has two options. The first is to add $t_1 \leq t_2 - 1$ as a new abstract variable and do another abstraction. Choosing this alternative is reasonable since it would undoubtedly result in a finer abstraction. The second choice is to formulate a *conjecture* and try to prove it. A conjecture (cf. *interactive abstractions* diagram, Fig. 3) is a property of the concrete system that, if proved correct, would eliminate some abstract counterexample. When it is not too difficult to come up with a relevant conjecture, this option should be preferred. This is because a proved conjecture usually eliminates more abstract counterexamples in further abstractions, and may significantly reduce the number of iterations to achieve a proof.

The feedback information can suggest a conjecture. In our example, the information "$t_1 \leq t_2 - 1$ *should not hold when* $pc_1, pc_2 = init$" obviously suggests the following conjecture: $\Box(pc_1 = init \wedge pc_2 = init \supset \neg(t_1 \leq t_2 - 1))$. But we choose to prove the following (stronger) property:

$$\Box(pc_1 = init \wedge pc_2 = init \supset t_1 = t_2). \tag{9}$$

Indeed, we notice that whenever both processes are at their *init* location, the stronger relation $t_1 = t_2$ (rather than $\neg(t_1 \leq t_2 - 1)$) holds ($t_1 = t_2$ is true initially, and any loop that goes back to the *init* locations increases both t_1 and t_2 by one). Thus, there is good chance that the prover alone will succeed in proving the conjecture. Indeed, (9) is proved by three iterations of invariant strengthening. We then use it as a proved in a second abstraction (also relative to $(a \leq t_1)$ and $(b \leq t_1)$). This time, the wishes are granted, and the prover can discharge the unproved subgoals and complete the proof.

An interesting conclusion can be drawn from this simple example. While the conjecture (9) can be proved by invariant strengthening, this is *not* the case for the

mutual-exclusion property (8) $\Box I$ itself. As shown in [BGP97], backwards analysis for this property does not converge, and hence (8) cannot be proved by $\widetilde{pre}$-invariant strengthening. Therefore, by Theorem 4.4, mutual exclusion cannot be shown by $\widetilde{pre}$-abstraction either. Moreover, it is not difficult to prove that even an abstraction relative to any finite set of *sub-formulas* of $\widetilde{pre}$-images of I (such as the guards of the transitions) cannot show (8). The reason for this is that to prove (8) it is important to know when $t_1 = t_2$ holds, but the $\widetilde{pre}$-images of I express only weaker relations between t_1 and t_2. (In the example we have obtained this information by proving the conjecture (9) instead of the weaker $\Box(pc_1 = init \wedge pc_2 = init \supset \neg(t_1 \leq t_2 - 1))$ that was suggested by the feedback information.)

This demonstrates a typical use of *interactive abstractions*, in which the detailed feedback from the system together with moderate amount of user ingenuity yields the relevant abstractions and auxiliary invariants. This is in contrast to an ordinary abstraction or theorem proving process, in which the user usually has to invest much more effort to come up with suitable abstractions and auxiliary invariants.

5 Conclusion and Future Work

As an attempt to address the problem of the significant user ingenuity that is required to come up with appropriate auxiliary invariants or with suitable abstraction mappings, we have presented a new methodology called *interactive abstractions* integrating static analysis, theorem proving and abstractions techniques. The key features of the approach are

- It is *incremental*: each step is based on information obtained from failures of previous steps. When the iterative process terminates, it yields a proof or a counterexample.
- It is *goal-directed*: abstractions are guided by a subgoals in a failed proof attempt.
- It is *partially automatic*: each component performs an automatic task, the user chooses which component to invoke at each step and how to apply it.
- User input is *highly guided* by information provided by the system.
- It is *general*, in principle, and not dependent on a particular implementation of the components.

The current implementation of *interactive abstractions* consists of a number of loosely coupled components. We use PVS [ORS+95] as the theorem-proving component. We have written a hierarchy of PVS theories to formalize temporal logic, transition systems (in general), and to describe a specific transition system (that

is being analyzed) and properties of the latter such as auxiliary invariants found by static analysis and proved conjectures. For static analysis and abstraction we use the tool Invariant Checker [GS97]. We are currently implementing the forward/ backward trace analysis as a separate component to analyze finite traces of transitions systems with boolean, enumerated types, integers, and lists. This is enough to model, e.g., communication protocols, in which shared list-variables are used to the communication channels. Our intention is to experiment with the achieved version of *interactive abstractions* on larger case studies.

Acknowledgments. We wish to thank John Rushby and Natarajan Shankar for valuable comments, Sam Owre for lending us help with PVS, and Hassen Saidi for assisting us with the Invariant Checker.

References

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142-170, 1992.

[BGP97] T. Bultan, R. Greber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In *Proc. of the 9th Conference on Computer-Aided Verification, CAV '97*, LNCS 1254, pages 400–411.

[BL] S. Bensalem and Y. Lakhnech. Automatic generation of invariants. To appear in *Formal Methods in System Design.*

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Constructing abstractions of infinite state systems compositionally and automatically. In *Proc. of the 10th Conference on Computer-Aided Verification, CAV '98*, LNCS 1427, pages 319–331.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages, POPL '77*, pages 238–252.

[CCF+97] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Muñoz, C. Murthy, C. Parent, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual Version 6.1. Technical Report RT-0203, INRIA, July 1997.

[CU98] M.E. Colòn and T.E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Proc. of the 10th Conference on Computer-Aided Verification, CAV '98*, LNCS 1427, pages 293–304.

[DGG97] D. Dams, R. Gerth and O. Grümberg. Abstract interpretation of reactive systems. *ACM Transactions in Programming Languages and Systems*, 19(2):253-291, 1997.

[GM95] M. Gordon and T.F. Melham. *Introduction to the HOL system.* Cambridge University press, 1994.

[GS96] S. Graf and H. Saidi. Verifying invariants using theorem proving. In *Proc. of the 8th Conference on Computer-Aided Verification, CAV '96*, LNCS 1102, pages 196–207.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. of the 9th Conference on Computer-Aided Verification, CAV '97*, LNCS 1254, pages 72–83.

[H91] G.J. Holzmann. *Design and validation of communication protocols.* Prentice Hall, 1991.

[HPR97] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[HS96] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe, FME '96*, LNCS 1051, pages 662–681.

[LPY97] K. G. Larsen, P. Petersson, and W. Yi. UPPAAL: Status & Developments. In *Proc. of the 9th Conference on Computer-Aided Verification, CAV '97*, LNCS 1254, pages 456–459.

[ORS+95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107-125, 1995.

[SUM96] H.B. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. In *Proc. of the 8th Conference on Computer-Aided Verification, CAV '96*, LNCS 1102, pages 208–219.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399